

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

AD A 216 404

(4)

## DTIC DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified			1b. RESTRICTIVE MARKINGS	
2a. SECURITY CLASSIFICATION AUTHORITY JAN 3 1990			3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.	
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE			4. PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TR-466	
5a. NAME OF PERFORMING ORGANIZATION MIT Lab for Computer Science		5b. OFFICE SYMBOL (If applicable)		5. MONITORING ORGANIZATION REPORT NUMBER(S) N00014-83-K-0125; N00014-89-J-1988
6a. ADDRESS (City, State, and ZIP Code) 545 Technology Square Cambridge, MA 02139		7a. NAME OF MONITORING ORGANIZATION Office of Naval Research/Dept. of Navy		
6b. ADDRESS (City, State, and ZIP Code) 1400 Wilson Blvd. Arlington, VA 22217		7b. ADDRESS (City, State, and ZIP Code) Information Systems Program Arlington, VA 22217		
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/DOD		8b. OFFICE SYMBOL (If applicable)		9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Blvd. Arlington, VA 22217		10. SOURCE OF FUNDING NUMBERS		
		PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.
				WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) Cache for Multi-Threaded Processors on a Split-Transaction Bus				
12. PERSONAL AUTHOR(S)				
13a. TYPE OF REPORT Technical		13b. TIME COVERED FROM TO		14. DATE OF REPORT (Year, Month, Day) 1989 November
15. PAGE COUNT 26				
16. SUPPLEMENTARY NOTATION				
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)	
FIELD	GROUP	SUB-GROUP	Multiprocessor, multi-threading, split-transaction bus, cache, coherency, consistency proof, Multilisp	
19. ABSTRACT (Continue on reverse if necessary and identify by block number)				
<p>A multi-threaded processor has several sets of registers, and therefore can keep several tasks in a state of being ready to run. This ability to combine several independent instruction streams prevents such a processor from getting systematically blocked upon every cache miss involving a long memory access. Unfortunately, upon a cache miss, conventional caches remain unavailable for further processor requests until the cache miss is completely processed. This of course defeats the purpose of this kind of architecture, since memory accesses performed by the other threads might hit in the cache and therefore succeed. Instead, the processor stays idle. This article describes a cache architecture capable of servicing processor requests even while a memory access is currently being performed. For further efficiency reasons, this cache communicates with the memory via a split transaction bus. These two features increase substantially the amount of state information to be kept along with each cache entry, making the cache automaton and protocol quite complicated. We detail the kind of consistency provided by our cache, along with a proof of its validity. As</p>				
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT <input type="checkbox"/> DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION Unclassified	
22a. NAME OF RESPONSIBLE INDIVIDUAL Judy Little			22b. TELEPHONE (Include Area Code) (617) 253-5894	22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR

83 APR edition may be used until exhausted.  
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

U.S. Government Printing Office: 1985-507-047

Unclassified

90 01 03 052

19. cont.

very little theoretical support exists for this kind of proof, we also present a formalism that we developed in the course of this project, and which is suitable for expressing statements of consistency.



Accession For	
NTIS GRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

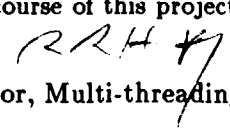
# Cache for Multi-Threaded Processors on a Split-Transaction Bus \*

Ingmar Vuong-Adlerberg <sup>†</sup>  
M.I.T. Laboratory For Computer Science  
545 Technology Square Cambridge Mass 02139  
and  
I.N.R.I.A  
Domaine de Voluceau Rocquencourt  
B.P. 105 78153 Le Chesnay Cedex, France

October 20th 1989



## Abstract

A multi-threaded processor has several sets of registers, and therefore can keep several tasks in a state of being ready to run. This ability to combine several independent instruction streams prevents such a processor from getting systematically blocked upon every cache miss involving a long memory access. Unfortunately, upon a cache miss, conventional caches remain unavailable for further processor requests until the cache miss is completely processed. This of course defeats the purpose of this kind of architecture, since memory accesses performed by the other threads might hit in the cache and therefore succeed. Instead, the processor stays idle. This article describes a cache architecture capable of servicing processor requests even while a memory access is currently being performed. For further efficiency reasons, this cache communicates with the memory via a split transaction bus. These two features increase substantially the amount of state information to be kept along with each cache entry, making the cache automaton and protocol quite complicated. We detail the kind of consistency provided by our cache, along with a proof of its validity. As very little theoretical support exists for this kind of proof, we also present a formalism that we developed in the course of this project, and which is suitable for expressing statements of consistency. — R.R.H. 

**Keywords:** Multiprocessor, Multi-threading, Split-transaction Bus, Cache, Coherency, Consistency proof, Multilisp.

---

\*This research was supported in part by the Defense Advanced Research Projects Agency and was monitored by the Office of Naval Research under Contract number N00014-83-K-0125.

<sup>†</sup>On leave from I.N.R.I.A. During 1989

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Overview of the cache</b>	<b>5</b>
2.1	The architecture of the cache . . . . .	6
2.2	Description of the cache protocol . . . . .	7
2.3	Correctness proofs . . . . .	11
2.4	Further improvements . . . . .	12
<b>3</b>	<b>Memory consistency, a theoretical framework:</b>	<b>13</b>
3.1	On formalizing memory transactions . . . . .	13
3.2	Defining cache consistency . . . . .	14
3.3	On Proving Cache Consistency . . . . .	17
3.4	Proof of lemma 3 . . . . .	17
<b>4</b>	<b>conclusion</b>	<b>19</b>
<b>5</b>	<b>Acknowledgments</b>	<b>20</b>
<b>6</b>	<b>mathematical appendix</b>	<b>22</b>

# 1 Introduction

In the past decade, major architectural achievements aimed at building fast processing elements were accomplished with *pipelining* and *parallelization*. The current expertise in machine architecture has a well-balanced knowledge about the design of highly pipelined processors and it appears that the major bottleneck in building fast computers is in providing memory systems that are capable of matching the speed of these fast processors. This problem has been extensively studied and is solved, in the case of single processor architecture, with the use of fast cache memories [11]. These cache memories appear to be efficient in matching processor speed as long as a memory access hits in the cache but do not improve anything when the access misses. In conventional sequential architecture the probability of a cache miss can be rendered arbitrarily small by building sufficiently large cache memories, and it turns out that in most applications, the speed ratio between the processor and the main memory is such that rather small caches (4 kilowords) are sufficient to cache up to 4 megawords of memory and to achieve a very high processor utilization (over 99 %).

The next step in making faster machines is naturally the use of several such processor/cache elements organized in a way that allows them to work in parallel. A big class of such parallel architectures is based on a model of computation that relies on the concept of shared memory. In this model, several processors work concurrently on the same set of data, using dedicated memory locations for synchronization purposes. It appears that the performance of such systems drops very rapidly as the number of processors grows for three main reasons:

- The need for synchronization variables implies that the corresponding memory locations cannot usually be cached. No matter how big the caches are, and how clever the cache management is, there will always be a minimum number of cache misses due to broadcasting of synchronization operations.
- The fact that data are shared means that a piece of data is very likely to be requested by different processing elements, and therefore increases the traffic between memory and caches.
- The communication medium is now also much more critical, since a given memory access now needs to compete with other processor requests, and therefore the communication delays between caches and memory become quite a bit higher.

To summarize, cache misses cannot be rendered arbitrarily rare any more, and in addition the delays induced by each cache miss are now substantially higher. To solve this new problem, three areas of computer architecture were extensively explored by the research community:

- Performances of efficient communication medium such as crossbar networks were studied and analyzed [9].
- Clever cache management schemes, such as the snoopy cache [4], were developed to reduce the traffic due to shared data.
- New processor architectures were developed using techniques such as pipelining and memory prefetching - or more recently multi-threading [12] - to allow the processor to work in parallel with a memory access, thereby diminishing the idle time due to non-cache accesses.

Interesting results have been obtained individually in each of these areas; nevertheless, when it comes to combining them into an efficient design they appear to be rather incompatible. The

simple and efficient snoopy cache works well only with a bus as a communication medium, while an efficient switching network requires a rather complicated directory-based cache management scheme [1].

In the Multilisp [5] project, we started by the design of the processor. We found the idea of multi-threaded architecture particularly elegant as an alternative to conventional pipelined design for the following two reasons:

- Instead of filling up the pipeline with instructions from the same stream, we fill it with instructions of different streams. In the conventional pipeline, if an instruction is delayed by a long memory access, the amount of work (instructions) available to the processor until it gets suspended waiting for the memory access to complete is no greater than the length of the pipeline. In a multi-threaded architecture, this is no longer true. The processor finds available work not only in its pipeline but also by switching to a different task, which means that the available stock of potentially executable instructions is now bounded above by the product of the pipeline's length and the number of ready-to-run task frames. It thereby achieves more parallelism between processor and memory.
- In a conventional design, these multiple threads would be executed by several processors, each one being attached to its own private cache. Of course, this avoids the problem of having several threads competing for the same cache slot and for that reason decreases the cache miss ratio. On the other hand, the effect of shared data being communicated between caches tend to increase the cache miss ratio. An intelligent sharing of cache space according to the needs of processes, combined with an effective management policy, can yield a higher hit ratio [13]. Furthermore, even with poor management of the competition for cache slots, higher processor utilization can be achieved [6].

Although multi-threaded processors are appealing for the previously presented reasons, current cache expertise does not allow actually taking advantage of their features. The purpose of our work was to study suitable caches, carefully isolating the problems that such an architecture induces. It turned out that the management of such caches requires a fairly complicated protocol to ensure coherency. In the course of our design research we therefore found the need for efficient ways of proving the correctness of an implementation. Although this problem is not new, it is only recently that people actually tried to specify complicated multi-processor architectures and deal with correctness proofs. We found that the classical formulations for various cache coherency properties [8] stated in *natural language*, were difficult to apply to complicated designs, particularly because there is no simple, accurate way to describe such designs using natural language. We therefore introduced a theoretical framework, in which one can express in a unified way all classical coherency properties. We then developed a methodology for proving statements within this framework, and showed how it is suitable for proving the correctness of cache coherency protocols.

In the section below we present the various problems that we had to solve for the design of our cache, and relate them to already known problems. We synthesize a variety of solutions into achieving our goal and give an overview of the resulting cache architecture. This is immediately followed by a section which describes precisely both the architecture and the cache management protocol. Next we state in what sense our design is correct and part of the proofs are detailed. This is followed by the presentation of our formalism along with an explanation of our motivations. In the last section of this paper, our proof of correctness based on this formalism is completed.

## 2 Overview of the cache

The ability to issue a memory request in advance, so that the time interval induced by the memory delay can be used by the processor to make progress on the execution of some other instructions, is common to both pipelined and multi-threaded processors. In conventional caches, as soon as the access generates a cache miss, the associated processor is held, preventing it from making any further progress on other instructions. This is of course rather unfortunate for the performance of the system, but has the advantage of yielding a quite simple cache design:

- There is no need for the cache to keep track of outstanding requests.
- There is no need for sophisticated management to handle the inconsistencies that may occur if a second request modifies the location involved in the outstanding memory access. All accesses are well serialized.

In the past few years, people have studied possible improvements to this situation with the introduction of buffered memory accesses [3]. Write and read buffers are used in order to make the cache *lock-up-free*. It was shown in [3] that buffering was particularly efficient for highly pipelined machines and for high miss rates. In a second paper [10] the same authors show that the consistency conditions are not that easy to meet, and conclude that access buffering can be allowed, but invalidation buffering (in snoop cache type of protocols) must be forbid. They propose a protocol that relies upon adjoining a new state to each entry, that allows or disables further accesses: If an entry is being updated, any further reads or writes by the processor must be rejected or deferred until the update is completed.

In our project we were not only interested in taking advantage of buffering, but also of a better communication medium, and therefore investigated the consequences of using a bus with split transactions. Such a bus has several advantages:

- A bus transaction is composed of an information transfer phase and a decoding phase. The decoding phase, which typically includes an access to a memory circuit, is the most time consuming operation and does not require the use of the bus itself. In a split-transaction bus it is possible to overlap the decoding phase with other transfers and thereby increase the bandwidth. This will actually increase the performance by a factor of at least two, therefore allowing a larger number of connected processors.
- A split-transaction bus allows a pipelined protocol and therefore allows pipelining the treatment of bus transactions, making the decoding phase in particular, non-critical. The cache hardware is then simpler and more efficient.

On the other hand, substantial complexity is added to the cache coherency protocol. Bus transactions are no longer atomic, and therefore the state of a line in a cache may change during the processing of the bus operation. For instance, if a cache  $C_1$  owns a dirty copy of the location  $X$ , and cache  $C_2$  requests this same piece of data, then  $C_1$  must write its copy back. The copy-back operation is not immediate but takes some cycles, during which any further requests to this same location traveling on the bus must be intercepted until the write-back is completed. This requires upgrading the entry state information to allow the cache automaton to behave properly. We have attempted to exhaust most possible variations on the cache protocol using a split transactions bus. It opens the door for many small optimizations that we carefully studied, reaching the conclusion that most of them were not worth the added complexity. The next subsection gives an overview of the cache architecture, and is followed by a paragraph that gives a precise description of our cache protocol, along with a list of the acceptable optimizations.

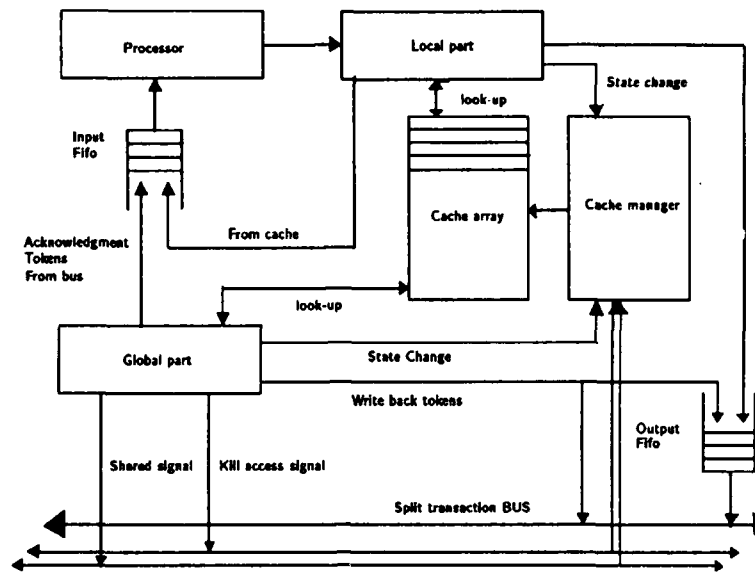


Figure 1: cache architecture block diagram

## 2.1 The architecture of the cache

A memory access issued by the processor is composed of an operation (Read, Write), an address, and possibly a piece of data. This is sent in one cycle to the cache as a token, and the processor then moves its attention to another instruction. This memory token may either hit in the cache, in which case an acknowledgment token is sent to the processor, or may miss, in which case a token is sent toward the memory via the bus. This token eventually comes back from the memory, appears on the bus, and is caught by the corresponding cache. It is then handled and a new token is produced and sent to the processor as an acknowledgment.

To accommodate this behavior we have developed a cache which, seen from the outside, has two input ports and two output ports (one pair on the processor side, and the other pair on the bus side) and a small memory array. Attached to the bus output port, a fifo queue allows tokens to be sent out without worrying about a possibly busy bus. Another small fifo buffer is put on the processor input path to accommodate the situation where, at a given cycle, a bus acknowledgment token arrives while the cache is responding to a request that hits in the cache memory. Each memory address  $X$  maps to a cache entry denoted  $\check{X}$  which records the address and data cached, as well as the current state of the cache entry and some additional bits e.g., to help with synchronization or garbage-collection. In the rest of the paper no assumption will be made about the memory-to-cache mapping; the cache can be a one-way associative cache as well as a many-way associative cache. Nevertheless we will use as a convention that  $\check{X}$  represents any address that maps onto the same entry as  $X$ , and  $\bar{X}$  is any such address that will require a cache replacement (in a one-way associative cache,  $\bar{X} = \check{X} - \{X\}$ ). The architecture and the operation of the cache itself can be described in terms of the following three modules:

- A *local part* in charge of processing requests performed by the processor, essentially by doing a cache lookup. As a result of this processing, this module may either send a



request to the *cache manager* or simply send a token to the fifo queue attached to the bus output port in order to perform a memory access.

- A *global part* which constantly watches the bus for acknowledgments or for requests from other processors. Each bus transaction therefore requires a cache lookup. This module may then either send a token directly onto the bus (bypassing the fifo buffer) along with the activation of the *Kill* signal (this happens when it detects a request to a datum that wasn't yet written back to the memory), or send a request to the *cache manager* to update an entry. This module may also activate a *Shared* signal, to notify the other processors that the data being acknowledged on the bus is shared.
- A *cache manager* which gets requests from both the *local part* and the *global part*. It performs a local operation on the cache array (Read, Write, State change) and if needed sends tokens to the *Local part* or the *Global part* which then propagate them to the processor fifo or to the bus fifo.

At any given clock cycle, the cache can get a request from both the processor and from the bus. In fact, at each active bus cycle the global part of the cache (which constantly observes the bus) must access the cache array in order to determine its action. Then it may modify the cache state (the previous lookup and this modification must occur atomically, for instance in the same cycle). In parallel with the treatment of bus transactions, the processor may read or modify an entry in the cache. These three actions that potentially occur within the same clock cycle must nevertheless be logically serialized at the cache array level and are therefore processed in the following order: First the cache entry is read by the snoopy unit, then a possible modification is reported by the same unit, finally the processor request is handled. The actual hardware may of course perform all this in parallel as long as the logical sequence is preserved.

## 2.2 Description of the cache protocol

In order to take proper actions upon *local part* and *global part* requests, the cache manager must maintain some information about the current consistency state of each cache datum. The conventional snoopy cache protocol requires the three following fundamental states to describe a given cache entry:

- Private and clean state (PC) which means that the data is held locally and is an exact copy of the corresponding location in memory. The associated cache is the only one that has a copy. The corresponding processor can therefore perform all the operations locally.
- Private and dirty (PD) which means that the data is held locally, and was modified by the associated processor. The copy is therefore not consistent with the one in memory. The associated cache is the only one that caches this location. The corresponding processor can still perform all the operations locally, but the cache is responsible for updating the memory as soon as another processor wants to read this location.
- Shared (S) which means that the data is held locally and in some other cache as well. The copy is also consistent with the corresponding memory location. The attached processor can still perform locally the *Read* operations but all *Writes* must be broadcast.

The fact that memory *Writes* and *Reads* are buffered requires adjoining the following state:

- Wait for bus transaction (WB) which means that the cache entry is being written to or read from the memory, and that a bus transaction was requested but the memory access has not yet traveled on the bus. Any further request to this cache entry, from the attached processor, is rejected until the corresponding acknowledgment token arrives.

Since we allow optimizing the memory accesses by observing and using the acknowledgment tokens which appear on the bus and match outstanding memory requests that are still sitting in the output buffer, we need the following state that indicates whether an access can be optimized or not.

- Wait for a memory acknowledgment (WA) which means that the memory access has traveled on the bus, but the memory acknowledgment wasn't yet observed on the bus. Any further request to this cache entry, from the attached processor, is rejected and the cache takes advantage of any acknowledgment token that appears on the bus and matches the address of this outstanding memory access.

The fact that transactions are split implies that write-backs are neither atomic nor instantaneous, therefore requiring this intermediate state.

- Wait for update acknowledgement (WU) which means that an update access was sent on the bus but the corresponding memory acknowledgment hasn't appeared yet. Any further request to this entry that appears on the bus must be cancelled (by activating the *kill* signal), and any request to this entry from the attached processor is rejected.

Finally, the obvious empty or invalid state (E) reflects the fact that at startup time a cache entry contains no valid data. Since our protocol uses no invalidations, the (E) state is never reentered. The above-listed states determine the actions to be taken by the cache manager upon occurrence of processor or bus events. The complete behavior of the cache is therefore fully specified by the description of its responses to each event or token. A token may induce either a state change (see figure 2), or an action such as the creation of a new token (see table). There are mainly two kinds of tokens:

- Requests from the attached processor, which are Reads or Writes to a given location, will be denoted by  $P_i : R@X$  or  $P_i : W@X$ , which stands for processor  $P_i$  (by convention  $P_i$  means the attached processor,  $P_j$  stands for any processor but the attached one and  $P_k$  stands for any processor) reading (respectively writing) at (to) location  $X$ . As a convention,  $X$  will represent the address that is cached,  $\bar{X}$  any address that causes a replacement of  $X$ 's cache entry, and  $\bar{\bar{X}}$  stands for any address that maps to the same cache entry as  $X$ .
- Tokens that travel on the bus, which can be either requests from another processor ( $P_j : R, W@X$ ) or write-back requests (updates) from another processor ( $P_j : U@X$ ) or, more likely, acknowledgment tokens  $P_k : Ra, Wa, Ua@X$  representing for a read/write/update acknowledgement tokens respectively. This notation is extended to reflect whether the *Shared* signal is activated or not upon the observation of the acknowledgment token. This is done by adding a ".s" (shared) or ".p" (private = not shared) modifier at the end of the token notation string.

We use this notation extensively in the rest of this paper. The cache manager is then completely specified with the transition diagram represented in figure 2 and the table of actions that summarizes the operations triggered by various events. Nevertheless, the following verbal description of the sequence of actions that take place for a given memory request helps in understanding the overall behavior:

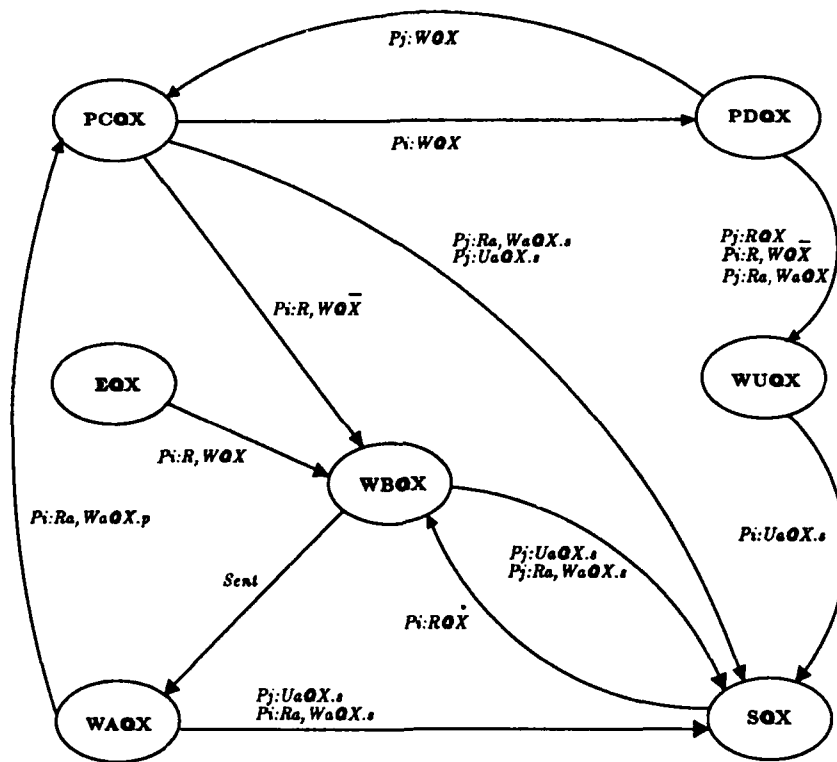


Figure 2: Cache automaton

- *Read hit:* When a processor performs a read operation that hits in its private cache, the cache simply sends to the processor an acknowledgment token that contains the desired data (the data can be either in PC, PD, or S state). If the entry is in an intermediate state (WA, WB, or WU) then the access is rejected, and the cache sends a rejection token to the processor, which will then reissue the request later.
- *Write hit:* If the entry is in a private state, then it is updated with the new value and the state is changed to PD if it were originally in the PC (clean) state. If the entry was in S (shared) state, then the write is broadcast by sending a token to the output fifo buffer and entering the WB (wait to obtain the bus) state. At this point if another acknowledgement token with the same address stamp is detected on the bus, the corresponding value updates the cache entry and an acknowledgment token is sent to the processor. The cache entry gets then into the S (shared state) again. If not, the request is eventually sent onto the bus and the WA state is entered. Two things may then happen: Either the acknowledgment token appears on the bus, or an update acknowledgment token appears. The second case occurs when the processor write was sent while an outstanding write-back was being processed. In this case <sup>1</sup> the processor which was responsible for this update, and which therefore was in the PD or WU state, has killed the regular acknowledgment token, and the update acknowledgment is to be taken into account instead. As a result, the write operation by our original processor will never be observed. In case the acknowledgment token does appear on the bus, any processor with a different number and which owns a copy of the same address in a stable state (PC, PD, S) must notify this cache by turning the shared signal on (several processors can do this concurrently). In this case, the cache entry gets into the S state. If the shared wire is not turned on then the cache entry gets into the PC state. Then an acknowledgment token is sent to the attached processor. Finally, if the write hit occurs during an intermediate state (WA, WB, WU) then a rejection token is simply sent to the processor.
- *Read miss:* In case the corresponding entry is in a dirty state, then the replacement cannot occur until the old cached data is written back. An update token is sent to the output fifo buffer, the WU state is entered and a rejection token is sent back to the processor so that it reissues its request later. While in the WU state, any request to the corresponding address that appears on the bus is cancelled, until the update acknowledgment is received and the S state entered. If the old data is in a PC or S state then the replacement can occur, a read token is sent to the output buffer to request the data from the memory and the WB state is entered. As in the previous case, if any acknowledgement token that carries the same address stamp appears on the bus, it is accepted as acknowledgment and the cache enters the S state. Otherwise, the token is sent on the bus and the WA state is entered. The rest of the protocol is as for the write hit.
- *Write miss: as for Read miss*

---

<sup>1</sup>Such a strange case is possible because of buffering: Processor  $P_i$  was in WB state, and meanwhile processor  $P_j$  got the same entry from the memory but in the PC state because  $P_i$ , not being in a stable state, hadn't turned the *shared* signal on. By the time the write token from  $P_i$  appeared on the bus,  $P_j$  got into the PD state.  $P_j$  is then responsible for updating the memory and it cancels  $P_i$ 's request, gets the next bus cycle and sends an update token bypassing its fifo buffer.

Event/State	E	WU@X	WA@X	WB@X	S@X	PC@X	PD@X
$P_i : R@X$	read token sent to output fifo	rejection token sent to processor			data fetched from cache and token sent to processor		
$P_i : W@X$	write token sent to output fifo	rejection token sent to processor			write token sent to output fifo	cache updated and token sent to processor	
$P_i : R, W@X$	read/write token sent to output fifo and cache updated with new address	rejection token sent to processor			read/write token sent to output fifo and cache updated with new address		rejection token sent to processor and update token sent to output fifo
$P_j : R@X$	ignored	kill sent	ignored				kill sent
$P_j : W@X$	ignored			kill sent	ignored		state changes but cache data not updated
$P_i : Ra, Wa@X$	impossible		cache updated and token sent to processor	impossible			
$P_i : Ua@X$	impossible	cache updated and shared signal sent	impossible				
$P_j : Ra, Wa@X$	ignored	kill sent	ignored	cache updated, shared signal sent			kill sent
$P_j : Ua@X$	ignored	impossible	cache updated		impossible	cache updated	impossible

Table: Actions taken by the cache upon bus and processor events.

### 2.3 Correctness proofs

Most currently published cache protocols are not formally proved because of their simplicity. Nevertheless our cache is substantially more complex than, for instance, the standard snoopy cache, and therefore one can think of many complex scenarios that might fool the automaton. We first made sure that the automata would never end up locked in a waiting state and for that purpose proved the following lemma:

**Correctness lemma 1 (Deadlock free automaton)** *For each transaction token sent to the output fifo buffer there is a unique acknowledgment token that travels on the bus (the automaton will therefore never stay in a waiting state)*

**Proof:** The proof of this statement is done by studying each type of bus transaction:

- $P_i : U@X$  is sent either directly onto the bus or to the fifo buffer, but it is guaranteed to be sent on the bus. It then can only be acknowledged by the corresponding update acknowledgement token.
- $P_i : R, W@X$  is sent by the processor to the output fifo buffer then (a) it does not get to the bus, in which case it was acknowledged by a token stamped with the same address (b) it does get to the bus but gets killed in which case there is another processor that started an update access and the corresponding acknowledgment token has not yet

appeared on the bus. An update token is always acknowledged, therefore the access will be acknowledged as well. (c) it does get to the bus and does get to the memory without being killed. the memory will then issue an acknowledgment token. This token is the only one that will be accepted by the automata: since the access was not killed, there was no outstanding updates by the time the request was sent on the bus. Therefore, any update acknowledgement that may appear on the bus correspond to subsequent update requests, and since the memory is a fifo server, will not appear on the bus until the regular acknowledgment token is sent (no unexpected acknowledgment will appear on the bus).

**Correctness lemma 2 (Correct state semantics)** *If a cache entry is in a private state (PC or PD) then no other cache can be in a stable state (PC, PD or S) at the same time (relatively to the global clock provided by the bus cycles).*

**Proof:** Trivial when considering the cycle at which the cache first got the data (which necessarily involved a bus access).

**Correctness lemma 3 (Sequential consistency)** *The system of caches is sequentially consistent.*

**Proof:** this proof which relies on the formalism presented in the next section, is given in section 3.4.

## 2.4 Further improvements

The cache that we described up to now supports a very simplified set of memory instructions. In particular it does not provide any *Test-and-set* type of operations, which makes it inefficient when supporting a programming language that uses storage locations for synchronization like Multilisp. Furthermore it does not support any kind of memory error handling, which is essential if a large system with a virtual address space is used. We have nevertheless investigated what would be involved in modifying our cache to work in a real machine:

- **Test-and-set instructions:** in Multilisp such synchronization is performed with the use of a *Full/Empty* bit that is attached to each memory location [7]. Standard Read/Write instructions are then extended to a variety of operations such as *read if location is full* or *write and leave location empty*. These operations can actually in most cases be treated as a simple Read or Write operation, but they no longer allow "on the fly" optimizations. The cache automaton should therefore be modified as if the WB state were projected onto the WA state, and the actions taken by the cache manager for Reads that side-effect a memory location should be treated as writes. The new system obtained is then write-ordered. In fact, with added complexity to the cache hardware, it turns out that "on the fly" optimizations can still be performed, but this is beyond the scope of this paper.
- **Virtual memory features:** For a virtual-address cache, an additional acknowledgment token must be introduced and recognized by the automaton in order to treat memory errors. The current automaton should still work as long as the processor that swaps memory pages in and out operates on the same bus with the same protocol provided that memory errors are treated by the cache automaton as normal acknowledgments.

### 3 Memory consistency, a theoretical framework:

For efficiency purposes, constraints regarding the ordering of transactions may get relaxed, leading to a more efficient use of the hardware (more parallelism) but getting also easily subject to protocol design errors, and therefore end with incorrect program execution. The purpose of this study is to review certain properties regarding the ordering of memory transactions in a multiprocessor system, which are commonly mentioned to satisfy the usual programming models. A formalism, using partially ordered sets, is proposed to define *cache coherence*, *sequential consistency*, and *write ordering* in a unified context. Along with this formulation, a theorem is provided and proved, aimed at helping the designer in proving that a given system satisfies any of the above properties.

#### 3.1 On formalizing memory transactions

Running a given parallel program on a multiprocessor system may lead to several different executions. An execution is characterized by its *Trace*, which is the partially ordered set (possibly infinite) of all memory transactions that are performed in the course of the run. Although we are rarely interested in a single trace, a legal execution is often defined as a class of traces, characterized by some constraints. These constraints have to do with both the sequential semantics of the programming language (which will define the behavior of individual processors), and its parallel semantics, which will impose constraints on the order of the operations in order to preserve the sequential aspect (data dependencies, sequential sections) of the execution of parallel programs. In this section we define several partial ordering relations on the set of all transactions performed by the system. These orders reflect the usual dependencies between memory instructions, within subsets of all memory transactions (order at a memory location ...). To achieve this goal, we need first to define the following objects:

- $\mathcal{T}$  is the set of all memory transactions performed by the system
- $\mathcal{R}$  is the subset of *Read* transactions :  $\mathcal{R} \subset \mathcal{T}$
- $\mathcal{W}$  is the subset of *Write* transactions :  $\mathcal{W} \subset \mathcal{T}$
- $P_i$  denotes a Processor/Cache ensemble.  $P_i \in \mathcal{P} = [0, Maxproc]$
- $A_j$  denotes a global memory address.  $A_j \in \mathcal{A} = [0, Maxadr]$
- $\mathcal{V}$  denotes the set of all possible values that can be stored in a memory location.
- $\Pi : \mathcal{T} \rightarrow \mathcal{P}$  is a function that canonically projects the set of transactions on the set of processors. (Each transaction  $X$  is performed by the unique processor  $\Pi(X)$ .)
- $\Delta : \mathcal{T} \rightarrow \mathcal{A}$  is a function that canonically projects the set of transactions on the set of Addresses. (Each transaction  $X$  is performed on the unique address  $\Delta(X)$ .)
- $\Sigma : \mathcal{T} \rightarrow \mathcal{V}$  is a function that canonically projects the set of transactions on the set of values (Each transaction  $X$  reads or writes the unique value  $\Sigma(X)$ .)
- $\Omega_{P_i} : \mathcal{T} \rightarrow \{0,1\}$  defines a family of characteristic functions that model the fact that a write transaction may or may not be *observed*<sup>2</sup> by the processor  $P_i$ .

---

<sup>2</sup>An observation is to be defined by the designer. Nevertheless it is assumed that all the transactions performed by a given processor are observed by this same processor.

We assume that within a processor  $P_i$  there is some kind of global synchronization (a single clock, or all memory acknowledgments arrive via a single stream, or there is a single instruction decode unit, etc, ...). Thus, even in a multi-threaded processor or in a processor with multiple memory ports, we assume that only a single instruction can be completed at a given clock cycle.  $P_i$  executes instructions in some sequence. Therefore, it can be assumed that there is a total order on the subset  $\Pi^{-1}(P_i)$  of all memory transactions performed by  $P_i$ . We denote this total order  $\leq_{P_i}$ .

A given location  $A_j$  in memory observes the requests performed on it in some order. For instance, in a system where a given address is always mapped to the same actual location (no cache for example), all the transactions involving  $A_j$  are serialized at the memory cell, and therefore are totally ordered. Nevertheless, this order is more often only a partial order (for instance where caches maintain copies of the contents of the location in memory). Let  $\leq_{A_j}$  be this order defined on  $\Delta^{-1}(A_j)$ . This order is likely to be consistent with  $\leq_{P_i}$  (If  $\tau_1$  is performed before  $\tau_2$  by processor  $P_i$  both at the address  $A_j$ , then  $\tau_1 <_{A_j} \tau_2$ ) but this is not always the case (for instance if the communication network does not preserve the order of messages).

Finally, there is an order derived from the programming language itself. The latter allows sequential sections, and therefore all the instructions within such a section are totally ordered. In a language such as Multilisp [5], there is a notion of parent and child tasks, which also impose a hierarchy and therefore a partial order (transactions performed by a child are subsequent, greater, than those from the parent task that were performed before the task creation. Transactions from the parent task that are subsequent to the child creation are greater than any transaction performed by the child). Let  $\leq_{prog}$  be this order defined on the entire set  $\mathcal{T}$ . The above discussion can be summarized as follow:

- $\leq_{P_i}$  is a total order on the transactions performed by the processor  $P_i$ .
- $\leq_{A_j}$  is a partial order on the transactions performed at the address  $A_j$ .
- $\leq_{prog}$  is a partial order on the transactions implied by the ordering of instructions within the program.

### 3.2 Defining cache consistency

In this section, various consistency properties are studied. Their usual definitions are interpreted and reformulated in different terms, in order to obtain a unified formulation of these properties. The general idea is that a consistency property is equivalent to the existence of some order on the set of all transactions resulting from a given execution.

**Definition 1 (Cache coherence)** *A system of caches is said to be coherent with respect to block A if each cache in the system observes all modifications of A in the same order.*

This definition given in [2] relies upon the meaning of "to observe" and therefore comes with the following definition: *A modification to a block A is observed by cache  $C_i$  if any subsequent read by processor  $P_i$  will return the newly written value.*

This latter definition assumes an obvious notion of "subsequent" among the transactions, which implies the existence of some global order. Along with the definition of cache coherence, this even appears badly formulated since cache coherence is about a notion of a global order. Moreover, the definition of cache coherence states that "all modifications" are observed which is not the case in most cache systems. As a matter of fact, in some cases each processor only observes the order of its own transactions. Thus, we prefer the following alternate definition,



which derives from a reordering of words in the previous formulation while remaining faithful to its spirit.

**Definition 2 (Cache coherence revisited)** *A system of caches is said to be coherent with respect to block A if for any execution, there is a total order  $\leq_A$  on the resulting set of all memory transactions performed at A that satisfies:*

- $\forall \tau_1, \tau_2 \in \Delta^{-1}(A)$  such that  $\exists i$  such that  $\tau_1 \leq_{P_i} \tau_2 \Rightarrow \tau_1 \leq_A \tau_2$  (The order is compatible with the order at each processor)
- $\forall r \in \Delta^{-1}(A) \cap \mathcal{R} \quad \Sigma(\sup \{w \in \mathcal{W} \cap \Delta^{-1}(A) \text{ such that } w <_A r\}) = \Sigma(r)$  (A read in this sequence returns the value of the most recent write)

In general this order is not unique. For instance in a two-processor coherent system, an execution may well give the following two traces<sup>3</sup> :

$P_1:W@X=2 \quad P_1:R@X=3$  (Processor 1)

$P_2:W@X=3 \quad P_2:W@X=3$  (Processor 2)

which may lead to several coherent total orders :

1/  $P_1:W@X=2 \quad P_2:W@X=3 \quad P_1:R@X=3 \quad P_2:W@X=3$

2/  $P_2:W@X=3 \quad P_1:W@X=2 \quad P_2:W@X=3 \quad P_1:R@X=3$

**Definition 3 (Sequential consistency)** *A system is sequentially consistent if the result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*

Sequential consistency is a property which relies upon the programming model, because it involves "program order" which is a notion that derives completely from the semantics of the programming language. Any program that fits into this model may lead to several different executions (some may terminate some may even not) but as long as the resulting trace can be ordered such that there is a sequential execution which leads to the same ordered trace, sequential consistency is satisfied. An alternate definition is:

**Definition 4 (Sequential consistency revisited)** *A system is sequentially consistent if for any execution, there is a total order  $\leq_{seq}$  on the resulting set of all memory transactions that satisfies:*

- $\forall \tau_1, \tau_2 \in \mathcal{T}$  such that  $\tau_1 \leq_{prog} \tau_2 \Rightarrow \tau_1 \leq_{seq} \tau_2$  (The order is compatible with the order imposed by the program)
- $\forall r \in \Delta^{-1}(A) \cap \mathcal{R} \quad \Sigma(\sup \{w \in \mathcal{W} \text{ such as } w \leq_{seq} r\}) = \Sigma(r)$  (A read in this sequence returns the value of the most recent write)

**Lemma 1** *Definition (3) and (4) are equivalent.*

**Proof :** (1) Definition (3) implies definition (4): Definition (3) implies the existence of a total order on the set of all instructions and therefore on the set of all memory transactions. This order is compatible with program order since it derives from a sequential execution of the program. For the same reason, the value returned by a read is the value Written by the last write in the sequential execution order.

<sup>3</sup>Notation:  $P:T@A=V$  means that processor  $P$  performs the transaction  $T$  (Read or Write) at the address  $A$  resulting with the value  $V$  (being Read or Written).

(2) Definition (4) implies definition (3): In order to prove that the existence of the order  $\leq_{seq}$  implies the existence of a sequential execution of the program, we will construct this execution by induction. We assume that instructions are also fetched from the same memory system as data although this is not necessary. The parallel run that leads to the totally ordered trace  $T$  starts with the system in some state. Since  $\leq_{seq}$  respects program order, the first read can perfectly be the first instruction executed in the sequential run. Suppose now that the sequential execution that leads to the same trace  $T$  up to the step  $k$  is built, let us prove that this execution can be completed to produce the transaction  $T_{k+1}$ . In the parallel run,  $T_{k+1}$  was executed by some processor  $P_i$ . Up to this point processor  $P_i$  was able to perform the same instructions and the same memory transactions as in the parallel run. In particular the task to which the transaction  $T_{k+1}$  belongs is in the same state as it was during the parallel run just before performing it (the state of the task which consists in the contents of some part of the memory and some registers like the Program counter, depends only on the sequence of data it has fetched from the memory up to the point where it is able to complete the next instruction). The new program counter is calculated using previously read data within the current task. Therefore the transaction  $T_{k+1}$  can be performed. It will be performed at the same address with the same value. Indeed, if this transaction is a write, both the address and the value written depend only on the current state of the task which is the same as for the parallel run. If this is a read transaction, the value read is the value that was written the most recently in the  $\leq_{seq}$  sequence. Since our sequential execution has already performed the transactions prior to step  $k$ , the value read will be the same as the one read during the parallel run. (One should note that this proof does not require that the programming model be deterministic).

Often, program order is the coalition of two orders : (1) the order of sequential threads (2) a data-flow order (synchronization variables, child tasks created by putting a token in some queue etc...). The latter order relies only upon the values involved in the memory transactions. Therefore the second condition that appears in definition (4) already guarantees that the data-flow order is respected. Thus, as long as each processor in the system executes instructions according to the order imposed by the sequential threads, program order is respected. For the case where program order is composed of the above-mentioned two orders, definition (4) can be reduced to:

**Definition 5** *A system is sequentially consistent if for any execution, there is a total order  $\leq_{seq}$  on the resulting set of all memory transactions that satisfies:*

- $\forall i \forall \tau_1, \tau_2 \in \Pi^{-1}(P_i)$  such that  $\tau_1 \leq_{prog} \tau_2 \Rightarrow \tau_1 \leq_{seq} \tau_2$  (The order is compatible with the order imposed by the program on the transactions performed by each processor)
- $\forall r \in \Delta^{-1}(A) \cap \mathcal{R} \quad \Sigma(\sup \{w \in \mathcal{W} \text{ such that } w \leq_{seq} r\}) = \Sigma(r)$  (A read in this sequence returns the value of the most recent write)

Sequential consistency does not imply cache coherence as it may appear at first. Indeed, nothing proves that the orders  $\leq_{P_i}$  of each processors transaction is respected by the global order  $\leq_{seq}$ . For instance in a multi-threaded processor system,  $\leq_{seq}$  may perfectly respect  $\leq_{prog}$  but not the  $\leq_{P_i}$ .

**Definition 6 (Write ordering)** *A system is write ordered if all caches observe all writes in the same order.*

**Definition 7 (Write ordering revisited)** *A system is write-ordered if for any execution, there is a total order  $\leq_{write}$  on the resulting set of all memory transactions performed that satisfies:*

- $\forall \tau_1, \tau_2 \in \mathcal{T}$  such that  $\exists i / \tau_1 \leq_{P_i} \tau_2 \Rightarrow \tau_1 \leq_{Write} \tau_2$  (The order is compatible with the orders of each processor)
- $\forall r \in \Delta^{-1}(A) \cap \mathcal{R} \quad \Sigma(\sup \{w \in \Delta^{-1}(A) \mathcal{W} \text{ such that } w \leq_{Write} r\}) = \Sigma(r)$  (A read in this sequence returns the value of the most recent write)

Assuming that each processor executes instructions in program order, then write ordering implies sequential consistency. The converse is not necessary true.

In this section we have assumed that observed order and processor order were the same. A stronger meaning can be given to "observed order", but it must always encapsulate processor order. Definition (2) and (6) can then be restated by replacing "processor order" by "observed order". The system will then be coherent or write ordered relatively to this new notion of observation.

### 3.3 On Proving Cache Consistency

The computer architect can usually easily prove that the basic processor respects program order. Further properties of consistency may be difficult to prove. The purpose of this section is to provide a technique for proving consistency properties. The computer architect, by designing the system, imposes some constraints on the way that memory transactions are performed. In particular, by putting bottlenecks (that will force the serialization of some messages) in various places, partial orders are imposed on transactions. By choosing an appropriate "consistent" set of partial orders, it is sometimes possible to build a total order that satisfies all of the partial orders. The following characterization provides a necessary and sufficient condition.

**characterization 1 (Consistent set of partial orders)** *Given a set  $\mathcal{T}$  of objects and a finite family  $\{<_i\}$  of partial orders, there is a total order that respects all of them if and only if any sequence :  $x_1 <_{i_1} x_2 <_{i_2} \dots <_{i_k} x_{k+1}$  (chain of objects where two adjacent elements are comparable by some order) does not loop (that is :  $x_{k+1} <_{i_{k+1}} x_1$  is impossible).<sup>4</sup>*

Often a partial order that is considered is a total order on the subset of objects to which it applies (for example, the order  $<_{P_i}$  applies only to the transactions performed by  $P_i$  but is a total order on this subset of  $\mathcal{T}$ ). If all the orders considered are such, then the following simpler characterization can be used :

**characterization 2** *If any sequence  $x_1 <_{i_1} x_2 <_{i_2} \dots <_{i_k} x_{k+1}$  where each order  $<_j$  appears at most once is not a loop then there is a total order that respects each partial order.*

The work of the designer is then reduced in both imposing some orders in his system and then finding a family of consistent partial orders. For instance, if program order is respected by each processor and if cache coherence is already proved, write ordering can be proved simply by showing that the family composed of  $\{<_{P_i}\}_i$  (orders on the transactions of  $P_i$ ) and  $\{<_{A_j}\}_j$  (order due to the coherence of the system relatively to each address  $A_j$ ) is consistent.

### 3.4 Proof of lemma 3

We first prove that there is a total order on the set of all memory transactions which is compatible with the order of the program:

---

<sup>4</sup>for a more precise formulation, see theorem (2).

- We can define a bus order  $\leq_{bus}$  using the order of the acknowledgment tokens. It is an order on all the transactions that were sent onto the bus but it can also be extended to all the transactions that were sent into an output fifo buffer. Indeed, if several transactions are acknowledged by the same token before they get onto the bus, they will simply be considered as being simultaneous.
- The order of each processor is defined as being the order of the acknowledgment tokens that arrive in its input buffer. This order can be extended to *local program* order. In fact, all write transactions that are acknowledged before they get onto the bus are never observed, and the write that corresponds to the bus acknowledgment is observed instead. We simply extend our order to insert the nonobserved write just before the observed one (no read operation could have been performed in between). We call this extended order  $\leq_i$  which, since each processor executes each thread in program order, is compatible with the program order.

It is easy to see that the bus order is compatible with each of these local orders ( $T_1 \leq_{bus} T_2$  and  $T_2 \leq_i T_1$  is impossible). Further it is also easy to see that any chain of three elements where each order appears only once cannot be a loop. Indeed, suppose we have  $T_1 \leq_1 T_2 \leq_2 T_3 \leq_3 T_1$  then by definition,  $T_1$  and  $T_2$  appeared in the same input buffer, and  $T_2$  and  $T_3$  both appeared in another input buffer. Therefore,  $T_2$  is a bus transaction, and so is  $T_3$  (because  $T_3$  and  $T_1$  appeared in a third input buffer). But the bus order can then compare these three transactions, and since it is a total order it implies that  $T_1 =_{bus} T_2 =_{bus} T_3$ . That is that all three transactions were acknowledged by the same bus token. In the exact same way we can prove that there is no loop of length  $k$ , therefore there is a total order on the set of all memory transactions that is compatible with each order  $\leq_i$ . This  $\leq$  order is therefore compatible with program order.

The second point that should be proved in order to demonstrate sequential consistency is that in the sequence defined by the above-constructed total order, each read returns the value of the most recent write. This is proved by analyzing each state in which the entry could have been at the time (relative to the bus clock) of the read. We proceed by cases:

- (PC) is the state of the entry: The automaton enters in the PC state only after a bus transaction. The value read is then the value that was written in the cache either by the most recent (relatively to  $\leq_i$ ) acknowledgment token ( $T_{a1}$ ) that came from the bus (in which case it was in the WA state just before), or by the attached processor (in which case it was in the PD state just before, and a token  $T_1 = P_j : W@X$  being detected on the bus forced the automaton back to PC).
  - Assuming the first case, between the moment that  $T_{a1}$  arrived and the time of read, no other acknowledgment stamped with the same address appeared on the bus (otherwise, the automata would not have stayed in the PC state). Therefore,  $T_{a1}$  is also the most recent acknowledgment relatively to  $\leq$  order. The lemma (2) proves that relatively to the bus clock no other processor has written to this address since  $T_{a1}$  has traveled on the bus. The current read then reflects the last write ( $T_{a0}$ ) in bus order (since the memory acts as a fifo). Suppose that some other processor has performed a write locally between  $T_{a0}$  and  $T_{a1}$  relatively to  $\leq$  putting its automata into the (PD) state. If the request token  $T_1$  that corresponds to  $T_{a1}$  was sent after this event then the write back would have worked because  $P_j : R@X$  would have been detected and  $T_{a0}$  would not be the most recent write. If  $T_1$  was sent before and  $T_{a1}$  sent after the local write, then the write back would have also worked because  $P_j : Ra, Wa@X$  would have been detected.

- Assuming the second case. the value read would be the most recent one relatively to  $\leq_i$ , since the last recorded modification  $W_0$  was performed by the attached processor itself. In any case, as previously, the last acknowledgment token  $Ta_1$  in bus order stamped with the corresponding address is the one that put the automata into the PC state before it went into the PD state. At the corresponding cycle no other processor had a copy of this location, therefore any subsequent foreign writes would have appeared on the bus. The only write that appeared on the bus (which is responsible for the transition PD to PC) is not yet acknowledged, therefore the attached processor is the only one that could have written the location since  $Ta_1$  in  $\leq$  order.  $W_0$  is therefore the most recent write in  $\leq$  order.
- (PD) is the state of the entry then, the proof is exactly as for the second case of the PC state.
- (S) is the state of the entry, then the value read is the one that was recorded in the cache by the last bus acknowledgment token  $Ta_1$  in both bus order and processor order (the automata can enter the S state only after such a token). This token could have been either an update acknowledgment token or a regular acknowledgment token.
  - Assuming the first case,  $Ta_1$  would then be the last acknowledged write in bus order. Suppose that some other processor  $P_j$  made a local write  $W_0$ . In  $\leq_j$  order  $W_0$  was preceded by a bus acknowledgment token  $Ta_0$  that is responsible for  $P_j$ 's automata entering the PC state. If we have  $Ta_0 \leq Ta_1 \leq W_0$  then  $Ta_1$  would have put  $P_j$ 's automata into the S state and  $W_0$  would not be a local write. If  $Ta_1 \leq Ta_0$  then  $Ta_1$  would not be the last acknowledged token in bus order. Therefore  $Ta_1$  is also the last write in  $\leq$  order.
  - Assuming the second case, and  $Ta_1$  being the last acknowledged read token in bus order (see above if it were a write token). The above proof shows that no local write can be between  $Ta_1$  and the current read in  $\leq$  order. Further at  $Ta_1$ 's bus cycle, another processor  $P_j$  had detected the token on the bus and since the shared line was activated, was in a state PC or S.  $P_j$ 's automata has entered the state PC or S upon a bus acknowledgement token  $Ta_0$  that is prior to  $Ta_1$  in bus order. We can assume that this is the most recent token stamped with the same address prior to  $Ta_1$ , otherwise we would have just considered the corresponding processor. If  $Ta_0$  is a write acknowledgment token, it is obviously the most recent acknowledged write in  $\leq$  order (the proof is similar to what was just done above).  $Ta_1$  would therefore report the value obtained from the memory which is the one written by  $Ta_0$ , the proof is then complete in this case. If  $Ta_0$  is a read acknowledgment token, then we can finish our proof by induction: Suppose that we have proved that any read prior to the current one in  $\leq$  order returns the value of the most recent write. Then we can apply the induction to  $Ta_0$  and we know that  $Ta_0$  gets its value from the memory. Since we already know that there is no writes between  $Ta_0$  and the current read in  $\leq$  order, we are sure that  $Ta_1$  returns also the same value as  $Ta_0$  which is the last written value.

## 4 conclusion

We have shown in this paper how the classical Snoopy cache protocol can be extended to accommodate the needs of a multi-threaded processor connected to the memory via a split-

transaction bus. This was achieved by using buffered accesses which required adjoining one state to the classical automaton, to indicate to the requesting processes that an access is currently being performed. Further, the non-atomicity of the transactions on the bus required another state to safely perform the write-back operations. Finally, for the sake of optimization we have added another state that allows certain buffered accesses to be acknowledged without being even sent on the bus simply by taking advantage of the other transactions passing on the bus. Our contribution to cache protocols resides in having studied this particular combination of processor/memory requirements, and to our knowledge this is the first published work that addresses the problem of efficient caching on split-transaction buses. Nevertheless, this quite specific cache can itself be rather easily extended to work with many communication media that use non-atomic transactions. In fact, in the course of our project we have adapted our cache automaton to work with a communication medium that is a combination of an interconnection switching network and a split-transaction bus (for broadcasts). In the latter case, part of the state information was kept in the memory (a very simple case of a directory scheme).

It appeared that the inflation in the number of automaton states, combined with an increased number of transition-triggering events (due to non-atomic transactions) made the whole design fairly complex, and therefore vulnerable to many errors. The need for satisfactory proofs, particularly to convince ourselves that our system achieved reasonable consistency, lead us to reexamine the problem of cache coherency. We found that the classical statements of consistency formulated in natural language were subject to many misinterpretations as the system to which they were applied got more complicated. This was a motivation for developing a more uniform formulation, based on partial orders on the set of memory transactions, that we used to re-state the classical definitions of various consistency properties. We found this formalism useful for understanding how to serialize the operations in our design, and it turned out to be a valuable guide for proving that our system was sequentially consistent.

## 5 Acknowledgments

The research reported here has benefited greatly from discussions with Bert Halstead, Dann Nussbaum and Randy Osborne.

## References

- [1] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An Evaluation of Directory Schemes for Cache Architecture. In *Proc. 15th Intl Symposium on Computer Architecture*, 1988.
- [2] J. K. Archibald. *The Cache Coherence Problem in Shared-Memory Multiprocessors*. University of Washington, 1986.
- [3] M. Dubois, C. Scheurich, and F. Briggs. Memory Access Buffering in Multiprocessors. In *Proc. 19th Intl Symposium on Computer Architecture*, 1986.
- [4] J.R. Goodman. Cache Memory Optimization to Reduce Processor/Memory Traffic. *Journal of VLSI and Computer Systems*, September 1987.
- [5] R.H. Halstead Jr. Multilisp: a language for concurrent symbolic programming. *IEEE Computer*, august 1986.
- [6] R.H. Halstead Jr. Processor Architecture for Multiprocessors. *Unpublished memo, Parallel Processing Group, M.I.T. Lab. for Computer Science*, January 1985.

- [7] R.H. Halstead Jr. and T. Fujita. MASA: a Multithreaded Processor Architecture for Parallel Symbolic Computing. In *Proc. 15th Intl Symposium on Computer Architecture*, pages 443-451, 1988.
- [8] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE. Transactions on Computers*, C-28, September 1979.
- [9] Janak H. Patel. Analysis of Multiprocessors with Private Cache Memories. *IEEE. Transactions on Computers*, C-31:296-304, April 1982.
- [10] C. Scheurich and M. Dubois. Correct Memory Operation of Cache-Based Multiprocessors. In *Proc. 14th Intl Symposium on Computer Architecture*, 1987.
- [11] A.J. Smith. Cache Memories. *ACM Computing Surveys*, 14-3:473-530, September 1982.
- [12] B.J. Smith. A Pipelined Shared Resource MIMD Computer. In *Proc. International Conference on Parallel Processing*, 1978.
- [13] P.C. Yeh, J.H. Patel, and E.S. Davidson. Shared Cache for Multiple-Stream Computer Systems. *IEEE. Transactions on Computers*, C-32:38-47, January 1983.

## 6 mathematical appendix

Given a countable set on which several partial orders are defined, the following section gives a sufficient condition, that guarantees the existence of a total order respecting all the partial orders.

**Definition 8** Let  $\mathcal{X}$  be a set and a **thread** be a pair  $(\mathcal{D}, \leq)$  where  $\mathcal{D}$  is a subset of  $\mathcal{X}$  and  $\leq$  is a total order on  $\mathcal{D}$ .  $\leq$  has a canonical extension as a partial order on  $\mathcal{X}$ , and  $\mathcal{D}$  will be called the domain of  $\leq$ .

Let  $\mathcal{X}$  be a countable set, on which a finite number  $k + 1$  of threads  $(\mathcal{D}_i, \leq_i)$  are defined. We will assume that  $\mathcal{X} = \bigcup_{i \in [0, k]} \mathcal{D}_i$

**Definition 9** A **chain** in  $\mathcal{X}$  is a finite enumeration <sup>5</sup>  $C : [0, m] \rightarrow \mathcal{X}$  such that:

$$\forall i \in [0, m-1] \exists j \in [0, k] \text{ such that : } C(i) \leq_j C(i+1)$$

$\mathcal{C}_m(\mathcal{X})$  is the set of chains of length  $m + 1$ ,  $\mathcal{C}(\mathcal{X}) = \bigcup_{m \in \mathbb{N}} \mathcal{C}_m(\mathcal{X})$  is the set of all chains of  $\mathcal{X}$ .

**Definition 10**  $\mathcal{E}(\mathcal{X})$  is the set of all enumerations that respect chaining in  $\mathcal{X}$ , and therefore partial ordering. i.e. the set of all injections  $I : \mathcal{N} \rightarrow \mathcal{X}$  such that :

$\mathcal{N} \in \mathcal{P}(\mathbb{N})$  (power set of  $\mathbb{N}$ )

$\forall m \forall C \in \mathcal{C}_m(\mathcal{X}), C(0), C(m) \in I(\mathcal{N}) \Rightarrow I^{-1}(C(0)) < I^{-1}(C(m))$  (respect chaining)

The following properties will then be considered:

**Property 1 (No accumulation point)**  $\forall \mathcal{P} \subset \mathcal{X}$  where  $\mathcal{P} \cap \mathcal{D}_i$  is infinite,  $\mathcal{P}$  is not upper bounded relative to  $\leq_i$ .

$$\forall x \in \mathcal{D}_i \exists p_i \in \mathcal{P} \text{ such that } x <_i p_i$$

This property leads to the fact that  $\forall \mathcal{P} \subset \mathcal{X}$  where  $\mathcal{P} \cap \mathcal{D}_i \neq \emptyset$ , there is a smallest element in  $\mathcal{P}$  relative to  $\leq_i$  ( $\forall i \in [0, k] \exists p_i \in \mathcal{P}$  such that  $\forall p \in \mathcal{P} p_i \leq_i p$ ).

**Property 2 (Coherence)** A chain cannot be a loop.

$\forall m \in \mathbb{N} \forall C \in \mathcal{C}_m(\mathcal{X})$  such that:

$$\forall x \in [0, m-1] \exists j \in [0, k] / C(x) <_j C(x+1)$$

then:  $\forall l \in [0, k] C(m) \not\leq_l C(0)$

**Property 3 (Connectivity)**  $\mathcal{X}$  is either a finite set, or satisfies the following property of connectivity:

For any element  $x$  in  $\mathcal{X}$ , the set of elements to which  $x$  is not chained, is finite:

$\forall x \in \mathcal{X} \{y \in \mathcal{X} / \forall m \forall C \in \mathcal{C}_m(\mathcal{X}) \cap \mathcal{E}(\mathcal{X}) \text{ such that } C(0) = x \text{ then } C(m) \neq y\}$  is finite.

**Theorem 1** Let  $\mathcal{X}$  be a countable set covered by a finite family of threads  $\{(\mathcal{D}_i, \leq_i)\}_{i \in [0, k]}$  and satisfying properties (1), (2) and (3). Then there exists a total order on  $\mathcal{X}$  that respects all the partial orderings. This order defines a complete enumeration of  $\mathcal{X}$ .

<sup>5</sup> An enumeration is an injection which maps a subset of  $\mathbb{N}$  (set of natural numbers) onto a subset of  $\mathcal{X}$ .



**Proof:** Lemma (1), (2) and (3) (see below) prove this theorem.

One can canonically define an order  $\preceq$  on  $\mathcal{E}(\mathcal{X})$  that encapsulates all the partial orders  $\leq_i$  defined on  $\mathcal{X}$ :

$$\begin{aligned} \forall I, J \in \mathcal{E}(\mathcal{X}) \quad I \preceq J \text{ if and only if:} \\ - \text{Image}(I) \subset \text{Image}(J) \\ - \text{The enumeration } J \text{ respects the order defined by } I \\ (i < j \Rightarrow J^{-1}(I(i)) < J^{-1}(I(j))) \end{aligned}$$

**Lemma 1** *If the above properties (1), (2) and (3) are satisfied, then  $\mathcal{E}(\mathcal{X})$  is inductive<sup>6</sup>.*

**Proof:**

Let  $\mathcal{S} \subset \mathcal{E}(\mathcal{X})$  be a filtering subset

Let  $X_{\mathcal{S}} = \bigcup_{i \in \mathcal{S}} \text{Image}(i) \subset \mathcal{X}$

The following properties are true:

(a) There exists a total order  $<_{\mathcal{S}}$  on the set  $X_{\mathcal{S}}$  compatible with all enumerations of  $\mathcal{S}$

**Proof:**  $\forall a, b \in X_{\mathcal{S}} \exists e_a, e_b \in \mathcal{S} / a \in \text{Image}(e_a) \ b \in \text{Image}(e_b)$ . therefore  
 $\exists e \in \mathcal{S}$  such that  $e_a \preceq e$  and  $e_b \preceq e$  (because  $\mathcal{S}$  is a filtering subset) and  
 $\forall f \in \mathcal{S}$  such that  $a, b \in \text{Image}(f) : f^{-1}(a) < f^{-1}(b) \Leftrightarrow e^{-1}(a) < e^{-1}(b)$   
because :  $\exists g \in \mathcal{S}$  such that  $f \preceq g$  and  $e \preceq g$   
one can then define :  $a <_{\mathcal{S}} b \Leftrightarrow e^{-1}(a) < e^{-1}(b)$   
(this relation is obviously an order)

(b) For any subset  $X_s$  of  $X_{\mathcal{S}}$  there is a smallest element relative to  $<_{\mathcal{S}}$

**Proof:**  $\forall i \in [0, k]$  such that  $X_s \cap D_i \neq \emptyset$ ,  $\exists m_i = \text{Min}_{<_i}(X_s \cap D_i)$  (by prop. (1))  
Because the set of  $m_i$  is finite and  $<_s$  is total  $\exists m = \text{Min}_{<_s}(m_i)$   
 $m$  is the smallest element of  $X_s$  relative to  $<_{\mathcal{S}}$  Indeed:  
 $\forall x \in X_s \exists i \in [0, k]$  such that  $x \in D_i \Rightarrow x >_i m_i \Rightarrow x >_s m_i \geq_s m$

Properties (a) and (b) allow us to define by induction an enumeration  $E$  of  $X_{\mathcal{S}}$  as follow:

$E(0) = \text{Min}_{<_s}(X_{\mathcal{S}})$  and  $E(k+1) = \text{Min}_{<_s}(X_{\mathcal{S}} - \{E(0), E(1), \dots, E(k)\})$

By construction, the enumeration  $E$  respects chaining.

It is obvious that:  $\text{Image}(E) \subset X_{\mathcal{S}}$  lets show that:  $\text{Image}(E) = X_{\mathcal{S}}$

Suppose  $\exists x \in X_{\mathcal{S}}$  such that  $x \notin \text{Image}(E)$  then  $\text{Image}(E)$  is infinite

(if it were finite, then because of the enumeration method  $\text{Image}(E)$  would be equal to  $X_{\mathcal{S}}$ )

By property (3)  $\exists y \in \text{Image}(E)$  to which  $x$  is chained.

Since  $\mathcal{S}$  is filtering  $\exists s \in \mathcal{S} / x, y \in \text{Image}(s)$

Since  $s$  respects chaining  $x <_s y$  and therefore  $x \in \text{Image}(E)$

$\forall s \in \mathcal{S} \ s \preceq E$  Indeed:

$\text{Image}(s) \subset X_{\mathcal{S}} = \text{Image}(E)$  and,

$\forall a, b \in \text{Image}(s)$  such that  $s^{-1}(a) < s^{-1}(b) \Rightarrow a <_s b$  and therefore:

$E^{-1}(a) < E^{-1}(b)$  (because of the construction of  $E$ )

Therefore  $\mathcal{S}$  is upperbound.

**Lemma 2**  $\mathcal{E}(\mathcal{X})$  has a maximal element.

<sup>6</sup> A partially ordered set  $\mathcal{E}$  is said inductive, if any filtering subset is upperbound. A subset  $\mathcal{S}$  of  $\mathcal{E}$  is said filtering if for any two elements  $x$  and  $y$  in  $\mathcal{S}$ , there is a third element  $z$  in  $\mathcal{S}$  that is greater than both  $x$  and  $y$ .

**Proof:** by Zorn<sup>7</sup> Theorem.

**Lemma 3** *There is a complete enumeration of  $\mathcal{X}$  which is maximal relative to the order  $\preceq$  defined on  $\mathcal{E}(\mathcal{X})$ .*

**Proof:** Let  $E$  be the enumeration obtained in the above lemma. We will prove that  $E(\mathbb{N}) = \mathcal{X}$ . Suppose  $\exists x_0 \in \mathcal{X}$  such that  $x_0 \notin E(\mathbb{N})$ . Then:

Property (1) guarantees that:  $\exists M / \forall i \in [0, k] \forall \alpha > M \quad E(\alpha) \not\prec_i x_0$

Moreover  $\exists m \in [0, M] / \forall i \in [0, k] \forall x \in E([0, M]) \cup \{x_0\} \quad E(m) \not\prec_i x$

**Proof:** If not, then (\*)  $\forall m / \exists i \in [0, k] \exists y \in E([0, M]) \cup \{x_0\} / E(m) >_i y$

We can then construct by induction an infinite chain as follows:

Let  $y_0$  be a random element of  $E([0, M]) \cup \{x_0\}$  and define  $y_n$  from  $y_{n-1}$  as:

$y_{n-1} >_{\alpha_n} y_n$  ((\*) guarantees the existence of  $y_n$ )

Since the set  $E([0, M]) \cup \{x_0\}$  is finite, this infinite chain has to be a loop.

This is in contradiction with the property (2).

We can now construct an enumeration of  $E([0, M]) \cup \{x_0\}$  that respect chaining, by induction:

Let  $E'(0)$  be such that  $\forall i \in [0, k] \forall x \in E([0, M]) \cup \{x_0\} \quad E'(0) \not\prec_i x$

And  $E'(n)$  be such that  $\forall i \in [0, k] \forall x \in E([0, M]) - E'([0, n-1]) \cup \{x_0\} \quad E'(n) \not\prec_i x$

(If there are several possibilities,  $E'(n)$  will be chosen to respect the order defined by  $E$ ) :

$E(\min(E^{-1}(\{y / \forall i \in [0, k] \forall x \in E([0, M]) - E'([0, n-1]) \cup \{x_0\} \quad y \not\prec_i x\})))$

$E'$  can be extended beyond  $[0, M+1]$  by taking the order defined by the enumeration  $E$  on  $E([M, \infty[)$ .  $E'$  respects chaining as well, and is strictly greater than  $E$ . Which is impossible.

**Theorem 2 (Theorem 1 bis)** *Let  $\mathcal{X}$  be a countable set covered by a finite family of threads  $\{(\mathcal{D}_i, \leq_i)\}_{i \in [0, k]}$  satisfying properties (1) and (2). Then there exists a total order on  $\mathcal{X}$  that respects all the partial orderings. This order defines a complete enumeration of  $\mathcal{X}$ .*

**Proof:**<sup>8</sup> The proof for this lemma is done by induction on the number  $k$  of threads involved. If there is only one thread, the lemma is trivial. Let suppose that theorem (2) is proved for  $k$  threads. The following two cases are to be considered:

(1)  $\forall i, j \in [0, k] \quad \mathcal{D}_i \cap \mathcal{D}_j$  is finite

We can then define for all  $i \in [0, k]$ ,  $m_i \in \mathcal{D}_i$  such that:  $m_i = \max_{<_i} \bigcup_{j \in [0, k], j \neq i} (\mathcal{D}_i \cap \mathcal{D}_j)$

Let  $\mathcal{D}_{>m_i} = \{x \in \mathcal{D}_i / x >_i m_i\}$  and  $\mathcal{D}_{\leq m_i} = \{x \in \mathcal{D}_i / x \leq_i m_i\}$

Since  $<_i$  is a total order on  $\mathcal{D}_i$  we have:  $\mathcal{D}_{>m_i} \cup \mathcal{D}_{\leq m_i} = \mathcal{D}_i$  and  $\mathcal{D}_{>m_i} \cap \mathcal{D}_{\leq m_i} = \emptyset$

Let  $X_{>} = \bigcup_{i \in [0, k]} \mathcal{D}_{>m_i}$  and  $X_{\leq} = \bigcup_{i \in [0, k]} \mathcal{D}_{\leq m_i}$  we have:

(a)  $X_{>} \cup X_{\leq} = X$  ( $X = \bigcup_{i \in [0, k]} \mathcal{D}_i = \bigcup_{i \in [0, k]} (\mathcal{D}_{>m_i} \cup \mathcal{D}_{\leq m_i}) \subset (X_{>} \cup X_{\leq})$ )

(b)  $X_{>} \cap X_{\leq} = \emptyset$ . Indeed:

If  $x \in X_{>} \cap X_{\leq}$  then  $\exists i, j \in [0, k]$  such that:  $x \in \mathcal{D}_{>m_i} \cap \mathcal{D}_{\leq m_j}$

Then  $x \in \mathcal{D}_i \cap \mathcal{D}_j$  and therefore  $x \leq_i m_i$  and  $x \leq_j m_j$

Which is in contradiction with the fact that  $x \in \mathcal{D}_{>m_i}$

<sup>7</sup>Any inductive set has a maximal element.

<sup>8</sup>A constructive proof for this theorem can also be done

Since  $X_{\leq}$  is finite, we can apply the theorem (1) to  $X_{\leq}$  with the family of threads  $\{(X_{\leq} \cap \mathcal{D}_i), <_i\}_{i \in [0, k]}$ . Therefore there is a total order, and an enumeration on  $X_{\leq}$  that respects all the partial orders. This finite enumeration can be completed with a standard diagonal enumeration<sup>9</sup> of  $X_{>}$ . (Thanks to the fact that the threads form a partitioning of  $X_{>}$ ) Because of property (1), this enumeration reaches all the elements of  $X_{<}$ .

(2)  $\exists i, j \in [0, k] \mathcal{D}_i \cap \mathcal{D}_j$  is infinite

*There is an enumeration of  $\mathcal{D}_i \cap \mathcal{D}_j$  that respects both  $<_i$  and  $<_j$ .*

Proof: Let  $d_{i,0} = \min_{<_i} (\mathcal{D}_i \cap \mathcal{D}_j)$  and  $d_{j,0} = \min_{<_j} (\mathcal{D}_i \cap \mathcal{D}_j)$

(property (1) ensures the existence of these both smallest elements)

Then  $d_{i,0} \leq_i d_{j,0} \leq_j d_{i,0}$  and therefore by property (2) :  $d_{i,0} = d_{j,0}$

Let  $d_{ij,0} = d_{i,0}$ , we can then define by induction the following enumeration:

$d_{ij,n+1} = \min_{<_i} ((\mathcal{D}_i \cap \mathcal{D}_j) - \{d_{ij,0}, d_{ij,1}, \dots, d_{ij,n}\})$

By property (1) again, this enumeration reaches any element of  $\mathcal{D}_i \cap \mathcal{D}_j$  and by construction it respects both orders. We will denote  $d_{ij,n}$  by  $d_n$ .

Let us define the following subsets of  $\mathcal{D}_{ij} = \mathcal{D}_i \cup \mathcal{D}_j$ :

$[d_l, d_{l+1}[_{<_i} = \{x \in \mathcal{D}_i / d_l \leq_i x <_i d_{l+1}\}$

$[d_l, d_{l+1}[_{i \rightarrow j} = \{x \in \mathcal{X} / \exists m \in \mathbb{N} / \exists C_m \in \mathcal{C}(\mathcal{X}) \cap \mathcal{E}(\mathcal{X}) /$

$C_m(0) \in [d_l, d_{l+1}[_{<_i}$  and  $C_m(m) \in [d_l, d_{l+1}[_{<_j}$  and  $x \in C_m([0, m])\}$

$[d_l, d_{l+1}[ = [d_l, d_{l+1}[_{<_i} \cup [d_l, d_{l+1}[_{<_j} \cup [d_l, d_{l+1}[_{i \rightarrow j} \cup [d_l, d_{l+1}[_{j \rightarrow i} \cup [d_l, d_{l+1}[_{i \rightarrow i} \cup [d_l, d_{l+1}[_{j \rightarrow j}$

(For any two elements of  $[d_l, d_{l+1}[$  any chain that connects them is also contained in  $[d_l, d_{l+1}[$ ).

*The above defined set  $[d_l, d_{l+1}[$  is finite:*

Proof: By property (1),  $[d_l, d_{l+1}[_{<_i}$  is finite. Let us show that the set of chains which respect the orders and end in  $[d_l, d_{l+1}[_{<_i}$  is finite. Let  $x_0$  be an element of  $[d_l, d_{l+1}[_{<_i}$ .

If the set of chains ending at  $x_0$  were infinite, then there would be an infinite set  $x_{0<}$  of elements of  $\mathcal{X}$  chained to  $x_0$ .

Therefore,  $\exists \alpha \in [0, k]$  such that  $x_{0<} \cap \mathcal{D}_\alpha$  is infinite, and consequently  $x_0$  is an accumulation point. This is in contradiction with property (1). This proves that the set  $X_{0<}$  is finite and so are  $[d_l, d_{l+1}[_{i \rightarrow j}$  and  $[d_l, d_{l+1}[$ .

$\mathcal{D}_{ij} \subset \overline{\mathcal{D}_{ij}} = \bigcup_{l \in \mathbb{N}} [d_l, d_{l+1}[$

*Properties (1), (2) and (3) are satisfied by  $\overline{\mathcal{D}_{ij}}$  with the family of threads  $\{(\overline{\mathcal{D}_{ij}} \cap \mathcal{D}_\alpha), <_\alpha\}_{\alpha \in [0, k]}$*

Proof: Properties (1) and (2) are directly derived from the hypotheses.

Let  $x \in \overline{\mathcal{D}_{ij}}$ ,  $\exists \alpha$  such that  $x \in [d_\alpha, d_{\alpha+1}[$

Then  $x$  is chained to any  $y \in [d_\beta, d_{\beta+1}[$  where  $\beta > \alpha$

( $x$  is chained to  $d_{\alpha+1}$  which is chained to  $d_\beta$  which is chained to  $y$ )

Therefore the set of elements to which  $x$  is not chained is included in  $\bigcup_{l \leq \alpha} [d_l, d_{l+1}[$  which is finite. This proves property (3).

<sup>9</sup>We start by numbering the first element of each thread, and then the second element of each thread etc, ...

Theorem (1) ensures the existence of a total order  $<_{ij}$  on  $\overline{\mathcal{D}_{ij}}$  that respects all partial orders and which defines an enumeration. In order to bring the proof to completion we now need to apply the induction hypothesis to the set  $\mathcal{X}$  with the threads  $\{(\overline{\mathcal{D}_{ij}}, <_{ij}), (\mathcal{D}_l, <_l)_{l \in [0, k] - \{i, j\}}\}$ . Property (1) is satisfied by hypothesis for  $\mathcal{D}_l \forall l \in [0, k] - \{i, j\}$ , but also for  $(\overline{\mathcal{D}_{ij}}, <_{ij})$  because of the enumeration defined by the order  $<_{ij}$ .

*Property (2) stands for  $[\mathcal{X}; \{(\overline{\mathcal{D}_{ij}}, <_{ij}), (\mathcal{D}_l, <_l)_{l \in [0, k] - \{i, j\}}\}]$ .*

**Proof:** Suppose there is a loop  $C_m$ . One can assume that  $C_m$  has the smallest possible length. Then, each order relation  $<_\alpha$  appears only ones in the chain. Indeed:

If  $\exists k, l \in [0, m], k < l$  such that  $C_m(k) <_\alpha C_m(k+1)$  and  $C_m(l) <_\alpha C_m(l+1)$

(a) if  $C_m(k) \leq_\alpha C_m(l+1)$  Then the chain obtained from  $C_m$  by removing the elements  $C_m(k)$  to  $C_m(l+1)$  is also a loop and is shorter

(b) if  $C_m(k) >_\alpha C_m(l+1)$  Then the chain obtained from  $C_m$  by keeping only the elements  $C_m(k+1)$  to  $C_m(l+1)$  is also a loop and is shorter.

$<_{ij}$  is involved in the loop, otherwise  $C_m$  would be a loop in  $(\mathcal{X}, (\mathcal{D}_i, <_i)_{i \in [0, k]})$ .

Therefore,  $\exists l \in [0, m]$  such that  $C_m(l) <_{ij} C_m(l+1 \bmod m)$

Since  $C_m(l), C_m(l+1) \in \overline{\mathcal{D}_{ij}}$  then

$\exists a, b / C_m(l) \in [d_a, d_{a+1}[$  and  $C_m(l+1) \in [d_b, d_{b+1}[$

(a)  $a = b$  is impossible, because the enumeration of  $\overline{\mathcal{D}_{ij}}$  which was constructed with theorem (1) respects all the chains that connect two elements of  $[d_a, d_{a+1}[$

(b)  $a < b$  is impossible, because then  $C_m(l)$  would be chained to  $C_m(l+1)$  via orders chosen from the original set of orders:  $(<_i)_{i \in [0, k]}$

Therefore  $C_m$  would be also a loop in  $(\mathcal{X}, (\mathcal{D}_i, <_i)_{i \in [0, k]})$ .

(c)  $a > b$  As case (b).

OFFICIAL DISTRIBUTION LIST

Director 2 copies  
Information Processing Techniques Office  
Defense Advanced Research Projects Agency  
1400 Wilson Boulevard  
Arlington, VA 22209

Office of Naval Research 2 copies  
800 North Quincy Street  
Arlington, VA 22217  
Attn: Dr. Gary Koop, Code 433

Director, Code 2627 6 copies  
Naval Research Laboratory  
Washington, DC 20375

Defense Technical Information Center 12 copies  
Cameron Station  
Alexandria, VA 22314

National Science Foundation 2 copies  
Office of Computing Activities  
1800 G. Street, N.W.  
Washington, DC 20550  
Attn: Program Director

Dr. E.B. Royce, Code 38 1 copy  
Head, Research Department  
Naval Weapons Center  
China Lake, CA 93555